Tarea API Productos (Flask)

Módulo 2: Fundamentos de Backend con Python

Alumno: Iñigo Uribarri Parada

URL: https://github.com/iuribarri2023/UEM_02_Python-Product_API

1. Introducción y objetivos del trabajo

Este documento describe de forma ordenada y entendible el funcionamiento de una pequeña API desarrollada en Flask que gestiona productos y pedidos para una tienda en línea. El objetivo es explicar cómo está organizado el proyecto, que tecnologías se utilizan, como se almacenan los datos en estructuras de datos (BST para productos y lista enlazada para pedidos) y como se puede probar la API mediante Swagger/OpenAPI

2. Arquitectura del proyecto

El proyecto separa la lógica principal, configuración, estructuras de datos, repositorios y esquemas de validación

2.1 Estructura de carpetas

Los archivos y directorios principales son:

- main.py: punto de entrada; crea la aplicacion Flask con create_app y la arranca
- app/__init__.py: factoría de la app; carga configuración, inicializa Flask-Smorest, registra el blueprint de pedidos/productos y define rutas basicas "/" y "/health"
- app/config.py: configuración general y metadatos OpenAPI
- app/extensions.py: instancia la Api de Flask-Smorest y otros componentes compartidos.
- app/api/orders.py: endpoints REST para productos y pedidos
- app/services/data_structures.py: implementación del árbol binario de búsqueda y la lista enlazada.
- app/services/order_service.py: lógica de negocio de pedidos (resolucion de items, totales).
- app/repositories/product_repository.py: acceso a datos de productos en JSON usando BST en memoria.
- app/repositories/order_repository.py: acceso a datos de pedidos en JSON usando lista enlazada en memoria.
- app/schemas/: esquemas Marshmallow para validar entradas y salidas de productos y pedidos.
- data/products.json y data/orders.json: archivos JSON que actuan como base de datos simulada.

2.2 Dependencias principales

Las dependencias clave (definidas en requirements.txt) incluyen:

- Flask: framework principal para construir la API.
- Flask-Smorest: gestión de blueprints, validación y generación automática de documentación OpenAPI.
- Marshmallow: definición de esquemas y validación de datos de entrada y salida.
- python-dotenv: carga de variables de entorno desde archivos .env durante el desarrollo.

3. Estructuras de datos: BST y lista enlazada

La API usa estructuras explicitas para cumplir los requisitos de eficiencia y modelado

3.1 Productos en arbol binario de busqueda (BST)

En product_repository se mantiene un BinarySearchTree en memoria:

- Cada nodo almacena un producto identificado por su id.
- Las operaciones de lectura (GET) usan tree.find y tree.inorder; las inserciones (POST) usan tree.insert
- La persistencia se realiza guardando el recorrido in-order en data/products.json, garantizando consistencia entre el BST y el JSON.

3.2 Pedidos en lista enlazada

En order_repository se mantiene un OrderLinkedList en memoria.

• Cada nodo representa un pedido completo con varios productos (items con product_id, cantidad y metadatos del producto).

• Las operaciones GET/POST/PATCH/DELETE recorren y modifican la lista (find, append, update, delete).

• Las mutaciones se persisten exportando la lista a data/orders.json mediante to_list().

4. Exposicion de la informacion API (Flask + Flask-Smorest)

La API se organiza en un único blueprint (app/api/orders.py) que agrupa productos y pedidos. Flask-Smorest genera la documentación OpenAPI y Swagger UI.

Endpoints principales:

- Productos:
  - GET /v1/products: lista productos (recorrido in-order del BST).
  - GET /v1/products/{id}: obtiene producto por ID desde el BST.
  - POST /v1/products: crea producto (name, price, stock, description).
- Pedidos:
  - GET /v1/orders: lista pedidos (recorrido de la lista enlazada).
  - GET /v1/orders/{id}: obtiene pedido por ID.
  - POST /v1/orders: crea pedido con items [{ product_id, quantity }]; resuelve productos, calcula totales y almacena el pedido en la lista enlazada.
  - PATCH /v1/orders/{id}: actualiza cliente o items recalculando el total.
  - DELETE /v1/orders/{id}: elimina un pedido de la lista enlazada

5. Gestión de datos y almacenamiento

Se usa un archivo JSON para cada entidad como base de datos simulada.

- data/products.json: inicializado con productos de construcción (cemento, arena, ladrillos, vigas, yeso).
- data/orders.json: inicializado con pedidos de ejemplo que referencian esos productos; cada elemento es un nodo de la lista enlazada al cargarse
- Los repositorios leen los JSON al iniciar y mantienen las estructuras (BST/lista) en memoria; cualquier cambio se persiste nuevamente al archivo correspondiente.

6. Documentación y pruebas con Swagger / OpenAPI

- El esquema OpenAPI se publica en /openapi.json.
- Swagger UI está disponible en /docs.
- Los esquemas Marshmallow alimentan la documentación (paths, parámetros, cuerpos y respuestas).
- Rutas de utilidad: / (landing con enlaces a docs y health) y /health (estado básico).

7. Pasos para ejecutar el proyecto en local

- Instalar dependencias: pip install -r requirements.txt.
- Lanzar la aplicación: python main.py.
- Probar en navegador: http://localhost:5000/docs para Swagger, y las rutas /v1/products y /v1/orders.

8. Limitaciones, riesgos y posibles mejoras

Limitaciones actuales:

- Uso de archivos JSON: sencillo para aprender, pero no soporta concurrencia ni grandes volúmenes.
- La lógica de negocio asume que los productos existen al crear pedidos; no hay control de stock ni transacciones.

Posibles mejoras futuras:

- Sustituir JSON por una base de datos real con migraciones.
- Añadir pruebas automáticas y validaciones adicionales (por ejemplo, stock disponible).
- Incorporar autenticación/autorización para consumidores de la API.
- Mejorar logs y manejo de errores.

9. Conclusiones

Esta API demuestra cómo un proyecto pequeño en Flask puede combinar estructuras de datos (BST y lista enlazada), validación con Marshmallow, documentación automática con OpenAPI y persistencia simple en JSON. La arquitectura por servicios, repositorios, blueprints y esquemas facilita extender el proyecto hacia escenarios más robustos.